

PROGRAMMING EXPERT

FAQ

Q Which programming language is best, Visual Basic or C#?

A When Visual Basic .NET was introduced it was only slightly different from C#, but over time the two main .NET languages have evolved in different directions. C# has gained advanced features while Visual Basic has evolved into Visual Basic 2005, which tries to make everything easier. The two languages seem to have diverged enough to give us a real choice.

Many of the differences are superficial, though, and most of the so-called extra features in VB are available to C# programmers if they choose to use them. VB6 programmers will find it easier to get to grips with VB 2005 than with C#, but learning C# has advantages: it's similar to C++, Java, JScript and many other modern languages. This makes it easy to generalise what you learn.

Mike James

IT author, lecturer
and programmer

mike@computershopper.co.uk

Automating applications

Did you know you can add scripting to almost any application without needing access to its source code? Mike James shows you how, using messages to automate applications



There are lots of programs that, for one reason or another, have no scripting facilities. Some, like Outlook Express, don't support scripting because the developers want you to buy a more expensive product that does. In other cases, the reason is more to do with the time and effort required to add scripting support. Sometimes this doesn't matter too much, but when you find yourself using an otherwise good program in exactly the same way repeatedly, you can miss the ability to automate the task.

The good, and perhaps surprising, news is that you can add scripting to almost any application without needing access to its source code. This month's project demonstrates how to use basic Windows mechanisms to gain control of an application. The technique is based on knowing how Windows works, and it's surprisingly easy for such a powerful set of tools that have a lot of exciting uses.

AUTOMATING A TASK

To demonstrate how everything works, we'll automate a simple task in an application called VOBedit. VOBedit is free to download, and easy to find by typing its name into a search engine. It's a freeware program that can be used to edit the video VOB files found on a DVD. A VOB file is essentially an MPEG2 file with MPEG2 and audio interleaved or multiplexed. The simplest way to edit video is to de-multiplex the VOB files into a single MPEG2 video file and a single audio file, usually encoded in AC3. VOBedit can do this.

In order to de-multiplex VOB files in VOBedit, you have to click the Open button, manually select the first VOB file in the dialog box that appears, click Open, select the 'demux' button, specify what you want de-multiplexed and specify where the files are stored. This isn't difficult, but if you want to repeat the process with

several DVDs it can become particularly tedious – hence the need for an automated solution.

Although it's something that nowadays is well hidden beneath layers of software, Windows is a message-passing operating system. All the controls, buttons, text boxes and so on that you see are just specialised windows that communicate with one another by sending messages. When you click a button, for example, it generates a message that is passed to its parent window. The parent window performs an action in response to the click.

There are so many types of Windows message, though, and so many are generated in a typical system that the majority are simply ignored by the windows that receive them. Today, we think about interactions between components as events and write event handlers to deal with them, but the messages are still there and still doing the same job in the same way. If you're happy working with events, there's little need to delve into the underlying Windows messages, but if you can't achieve what you want using events, direct access to messages is easy to implement. It can also open up some remarkable abilities.

You can, for example, quite easily send a message from one application to another. This means a message can cross 'process boundaries', something that is very difficult to do by other means. You can, for example, send a message to a window, informing it that a button it hosts has just been clicked. Similarly, you can send messages that set or get any text stored in text boxes that belong to other applications running on your PC. This can be a very powerful tool.

SEND AND NOTIFY

The .NET framework has very limited support for messaging. Fortunately, there is nothing to stop you

using the original Windows messaging API via P/invoke. There are two key API calls that allow you to send messages to other applications: `SendMessage` and `SendNotifyMessage`. These must be defined at the start of your application as:

```
[DllImport("user32.dll", CharSet =
    CharSet.Auto,
    SetLastError = false)]
public static extern IntPtr
    SendMessage(
        IntPtr hWnd, uint Msg,
        IntPtr wParam, IntPtr lParam);
[DllImport("user32.dll", SetLastError
    = true,
    CharSet = CharSet.Auto)]
public static extern bool
    SendNotifyMessage(
        IntPtr hWnd, uint Msg,
        IntPtr wParam, IntPtr lParam);
```

Both use the same set of parameters to send a message, but they behave differently once the message has been sent. `SendMessage` waits until the target window has accepted and processed the message, whereas `SendNotifyMessage` just drops off the message into the message queue and returns at once. A message is specified by four parameters:

hWnd	The 'handle' of the window to which the message is to be sent
Msg	An unsigned 32-bit value indicating the type of message
wParam	A 32-bit value usually but not always specifying the sender of the message
lParam	A 32-bit value specifying other features of the message

A handle is just a way to keep track of objects across process boundaries. Every window has a window handle that identifies it uniquely. There are thousands of different message types available, but the good news is that we can do nearly everything we need with just a few of them.

LOADING THE APPLICATION

Before we get started on the task of passing messages, we need to create a running instance of the application we want to control. This can be most easily achieved using the `Process` class. Start a new C# Windows application project, then enter the definitions for the two API calls given earlier and three 'using' statements:

```
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Threading;
```

Also add a button titled Run VOB job. Clicking on this will start the application and run the automated task. We'll also need a global variable to keep track of the application, so add:

```
public Process VobEdit = new
    Process();
```

We can now add the code that creates the application into the button's click event handler:

```
private void button1_Click(object
    sender,
    EventArgs e)
```

```
{
    VobEdit.StartInfo.FileName =
        @"VobEdit.exe";
    VobEdit.Start();
    VobEdit.WaitForInputIdle(5000);
```

For simplicity, this assumes that the `VOBEdit.exe` program is stored in the same directory as the C# project, but you could use the complete path to an existing application instead. The `Start` method creates a new instance of the application and `WaitForInputIdle` waits for five seconds or until the application enters the Idle state. Once the application enters Idle, it has finished loading and is ready for use.

USING THE OPEN BUTTON

Now that the `VOBEdit` application is running we need to get its window handle so that we can send messages to it. The `Process` class automatically gets the application's main window handle when it starts:

```
IntPtr hWnd = VobEdit.MainWindow
    Handle;
```

All the standard Windows controls send a `WM_COMMAND` to let their parent window know that they have been clicked. The documentation reveals that the code for `WM_COMMAND` is 111 (in hexadecimal) and a suitable constant can be defined:

```
const uint WM_COMMAND = 0x111;
```

The third parameter of the message-sending call has to specify the control ID of the button that has been clicked. All controls have a control ID that never changes. This means that we need to discover the control ID of the Open button. To do so, you can use a utility such as `Spy++`, which is included with Visual Studio, or `Winspector`, which is included on this month's cover disc. By running `Winspector` you can discover the control ID of any control hosted by a given window, as

well as details of what messages are being passed between windows.

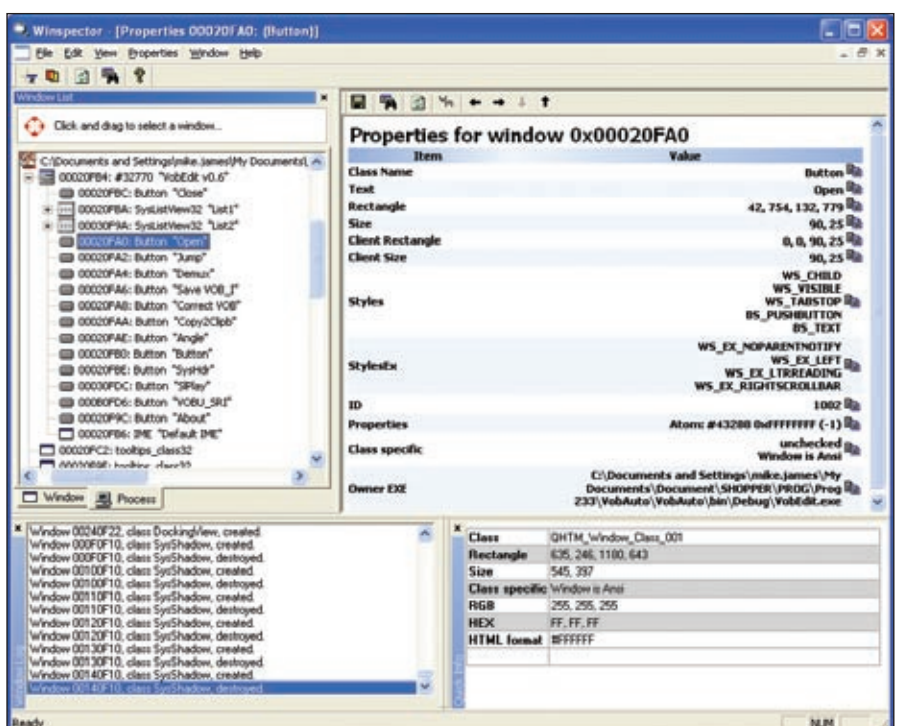
Using `Winspector`, we can see that the control ID of the Open button is 1002. Next we must know the window handle, which we need for the final parameter of our message. Unlike the control ID, the window handle changes each time the program is run. Fortunately, there is another API call that will find the current window handle of any given control ID. Add the following definition to the start of the program:

```
[DllImport("user32.dll")]
static extern IntPtr GetDlgItem(
    IntPtr hWnd, int nIDItem);
```

Now we can get the window handle and send the `WM_COMMAND` message to the main window:

```
IntPtr OpenBtnHwnd = GetDlgItem(hWnd,
    1002);
SendNotifyMessage(new
    HandleRef(VobEdit, hWnd),
    WM_COMMAND,
    new IntPtr(1002),
    OpenBtnHwnd);
```

Now if you run the program, you will see `VOBEdit` start, then the Open file dialog box will appear just as it does when you click the Open button. There are a few things to note about the way the message is sent. The first is that if you don't use the `HandleRef`, the `VOBEdit` process might be destroyed while you're working with it. If you use a `HandleRef`, it also means the `Hwnd` is passed to the API in a safe way. In most cases, you don't need to specify the window handle and the last parameter can be set to zero, but it's safer to pass it correctly. `SendNotifyMessage` also has to be used because the Open button creates a modal dialog box that pops up in front of the main window and freezes its message processing until the dialog is closed.



▲ Winspector reveals the control IDs for all the controls in the main window

COMPUTER STEP

Visual Basic Express in Easy Steps

★★★
£7.25



We really wanted to like this book. It's reasonably priced for a full-colour book, even if it is fairly short. Its presentation is excellent with lots of coloured icons marking hints, tips and warnings. Unfortunately, it is simply too short to explain such a big topic adequately, and when you start to read, rather than just look at the pictures, it's just a mess.

The book makes no attempt to confine itself to a subset of Visual Basic Express that's suitable for beginners. It covers simple things such as installing VB Express and creating simple programs that add two numbers together, but also advanced topics such as databases and XML. All the advanced topics are included at the expense of covering the basics well enough for the beginner to understand what is happening.

You might understand each of the topics covered, type in the examples and vaguely appreciate the ideas, but once you are left to develop your own programs, you will quickly realise how much you don't know. If all this wasn't enough, there is a chapter on using VBA to script Microsoft Office. Not only is VBA a different language, it's soon to be replaced by Visual Basic Express. This is a bit like finding a chapter on learning Latin tucked away inside a book on modern Italian.

This book is colourful, but ultimately useless.

SUMMARY

- ✓ Attractive format
- ✗ Tries to covers too much

SUPPLIER www.amazon.co.uk
ISBN 184078329X
PAGES 192

APRESS

Pro C# with .NET 3.0 Special Edition

★★★★
£27



Author Andrew Troelsen has been working on this book for some time and, while it might claim to be the first edition of a new book, it benefits from Troelsen's knowledge from writing many other books on similar subjects. This new book includes chapters on the new technologies about to be introduced with .NET 3.0, such as Linq, WPF and WCF. If you are already up to speed with .NET 2.0, much of the book will be unnecessary and you'd be better off buying something that covers only the new features to be introduced in .NET 3.0.

What's more, the text rarely departs from the well-beaten track of the documentation. It doesn't explore any of the possibilities of the technology, choosing instead to present the standard ideas. At least the presentation of the standard ideas has got better over time. It suffers from being extremely thick and unwieldy, though, and its bulk is exacerbated by many tables and details that you could easily find in the online documentation. However, it's so large that you're bound to find something helpful when you have a problem.

Troelsen's book rarely rises above the level of documentation. The official documentation is so bad that you could find the book useful, but this huge tome feels like an exercise in quantity rather than quality.

SUMMARY

- ✓ Covers .NET 3.0 technologies
- ✗ Regurgitates standard ideas

SUPPLIER www.amazon.co.uk
ISBN 1590598237
PAGES 1,250

If we used SendMessage, our application would wait until the dialog box was closed.

FINDING A DIALOG BOX

Next we need to find the window handle of the dialog box we've just opened. It's easy to find the child windows of a given window, but a dialog box is harder because it isn't a child window. It's an 'owned top-level' window. To find a dialog box, we have to write a function that examines each top-level window in turn until it finds one with the right characteristics.

To scan through all the top-level windows, we can use EnumWindows and its associated callback function, the definitions of which need to be added to the start of the program:

```
[DllImport("user32.dll")]
[return: MarshalAs(UnmanagedType.Bool)]
static extern bool EnumWindows(
    EnumWindowsProc lpEnumFunc,
    ref IntPtr lParam);
public delegate bool
    EnumWindowsProc(
        IntPtr hWnd, ref IntPtr lParam);
```

EnumWindows calls the callback function with the window handle in hWnd for each top-level window. The final parameter lParam is passed to the callback function and can be used to get back a result. The enumeration can be stopped when you find the window you are looking for by getting the callback to return false. The best way to use this API call is to create a new method that searches for a dialog box with a specified title and with the specified window as its Owner:

```
public IntPtr getDialog(IntPtr
    Owner,
```

```
String Caption,int timeout)
{
```

A timeout in milliseconds is included because we have to deal with the possibility that the dialog window we are looking for hasn't been created yet. In that case, we might need to try to find it more than once.

First, we must create the callback function. This is most easily done using the anonymous method facility introduced in .NET 2.0. This lets us define a delegate without first having to make a separate named function. Think of it as a direct or inline definition of the delegate object:

```
EnumWindowsProc enumProc = delegate(
    IntPtr handle, ref IntPtr pointer)
{
```

We need to test that the windows title matches the one we are looking for:

```
int length = GetWindowTextLength
    (handle);
StringBuilder wTitle = new
    StringBuilder(
        length + 1);
GetWindowText(handle, wTitle,
    wTitle.Capacity);
if (wTitle.ToString() == Caption)
```

If it does, we check that it is a dialog box by retrieving its class name, which for most dialogs is #32770. You can find the class name of any window using Spy++ or Winspector:

```
int max = 100;
StringBuilder classname = new
    StringBuilder(max);
```

```
GetClassName(handle, classname,
    max);
if (classname.ToString() ==
    "#32770")
```

If the class name proves that we have a dialog box, the final test is to check that its owner is the specified window:

```
IntPtr Parent =
    GetParent(handle);
if (Parent == Owner)
{
    pointer = handle;
    return false;
}
return true;
```

The GetParent API call returns either the owner or parent window depending on the type of window. This completes the callback function. All that remains is to use EnumWindows to use it:

```
DateTime end =DateTime.Now.
    AddMilliseconds(
        (double)timeout);
IntPtr DlgHwnd = IntPtr.Zero;
do
{
    Thread.Sleep(100);
    EnumWindows(enumProc, ref
        DlgHwnd);
}while(DlgHwnd==IntPtr.Zero &&
    end > DateTime.Now );
return DlgHwnd;
```




We need to use Sleep to give other threads time to open dialogs and complete other processes. To make all this work we need the following API function definitions:

```
[DllImport("user32.dll", SetLastError = true)]
static extern int GetClassName(
    IntPtr hWnd, StringBuilder
    lpClassName,
    int nMaxCount);
[DllImport("user32.dll",
    ExactSpelling = true,
    CharSet = CharSet.Auto)]
public static extern IntPtr
GetParent(IntPtr hWnd);
[DllImport("user32.dll", SetLastError = true,
    CharSet = CharSet.Auto)]
public static extern int
GetWindowText(
    IntPtr hWnd, [Out] StringBuilder
    lpString,
    int nMaxCount);
[DllImport("user32.dll", SetLastError = true,
    CharSet = CharSet.Auto)]
static extern int GetWindowTextLength
(IntPtr hWnd);
```

OPENING A FILE

With the getDialog method completed, we can now move on to use any dialogs we open. In this case, we want to use the Open File dialog box with the title "Open":

```
IntPtr OpenHwnd = getDialog(Hwnd,
    "Open", 5000);
```

We now need to enter text into the dialog's edit control. Winspector reveals that this control has a control Id of 1152. Its window handle is:

```
IntPtr EditHwnd =
    GetDlgItem(OpenHwnd, 1152);
```

We can use the WM_SETTEXT message to send a string to any edit control. The code for this is:

```
const uint WM_SETTEXT = 0xC;
```

We also need a slightly different definition of SendMessage to cope with a pointer to a string:

```
[DllImport("user32.dll", CharSet =
    CharSet.Auto,
    SetLastError = false)]
static extern IntPtr SendMessage(
    IntPtr hWnd, uint Msg,
    IntPtr wParam,
    [Out, In] StringBuilder
    lParam);
```

You can enter both definitions and the system will work out which to use in any particular case.

We could now just send the text, but even though the dialog and its edit control exist they could be in the middle of initialising. This would result in the text being written successfully, only to be blanked out again as the initialisation is completed. The correct way of dealing with this is to enquire as to the state of the dialog or set the text and check that it is indeed set (using a GetText message). However, a simpler solution is just to yield control to the other

threads in the system for a tenth of a second or more:

```
Thread.Sleep(100);
StringBuilder file = new
    StringBuilder(
        @"D:\VIDEO_TS\VTS_01_1.VOB");
IntPtr set = SendMessage(
    new HandleRef(VobEdit, EditHwnd),
    WM_SETTEXT,
    IntPtr.Zero, file);
```

The filename entered into the dialog box is the default name for the first VOB file in a video DVD. Finally, with the filename entered we can activate the Open button. The Open button has the control ID 1, and we can click it by sending it a WM_COMMAND message:

```
IntPtrDlgOpenBtnHwnd =
    GetDlgItem(OpenHwnd, 1);
SendMessage(new HandleRef(VobEdit,
    OpenHwnd),
    WM_COMMAND, new IntPtr(1),
    DlgOpenBtnHwnd);
```

This closes the dialog box and returns its results to the application.

DOING THE DEMUXING

We've told the application which file we want to process. The next task is to set the processing conditions and the location of the output file. To do this, we need to send a message that clicks the Demux button. This should be a similar task to that of operating the Open dialog box, but there's a problem. After the Open dialog is dismissed, the application goes off to find the file we specified and read its details. If we click the Demux button before it has finished this, the file open action is aborted and nothing works.

The application shouldn't allow this to happen. A more sensible way to work would be for it to disable the Demux button until it's ready to continue, but we can't change this as it's one of the idiosyncrasies of the VOBedit program. To get around the problem, we could tell our program to read the text from the edit control that displays the file details. Once this text has appeared, we know the application is ready so we can click the button. Alternatively, a simpler solution is to wait a while. We'll wait for a second before clicking the button and getting the handle of the dialog that appears:

```
Thread.Sleep(1000);
IntPtr DemuxBtnHwnd =
    GetDlgItem(Hwnd, 1005);
SendMessage(new
    HandleRef(VobEdit, Hwnd),
    WM_COMMAND, new IntPtr(1005),
    DemuxBtnHwnd);
Thread.Sleep(100);
IntPtr DemuxHwnd = getDialog(Hwnd,
    "Dialog", 5000);
```

One second should be enough for most users, but you might want to increase the delay slightly if your DVD drive spins up slowly.

With the Demux dialog now opened, we need to set the tick box controls for 'Demux all Video streams' and 'Demux all Audio streams'. These have the control IDs 1032 and 1031. To set the controls we need to use a new message BM_SETSTATE:

```
const uint BM_SETSTATE = 0xF1;
```

This sets the control if the value of wParam is 1, and unsets it if the value is 0. The message must be sent to each control's window:

```
IntPtr AllVidBtn =
    GetDlgItem(DemuxHwnd, 1032);
IntPtr AllAudBtn =
    GetDlgItem(DemuxHwnd, 1031);
SendMessage(new HandleRef(VobEdit,
    AllVidBtn),
    BM_SETSTATE, new IntPtr(1),
    IntPtr.Zero);
SendMessage(new HandleRef(VobEdit,
    AllAudBtn),
    BM_SETSTATE, new IntPtr(1),
    IntPtr.Zero);
```

Once the two tick boxes are set, we can send a message to click the OK button, closing the dialog box. The button has the control ID of 1:

```
IntPtr OkBtn = GetDlgItem(DemuxHwnd,
    1);
SendMessage(new HandleRef(VobEdit,
    DemuxHwnd),
    WM_COMMAND, new IntPtr(1), OkBtn);
```

REACHING THE FINISHING LINE

After our program dismisses the Demux dialog box, VOBedit will open a Save As dialog box. In this box, we need to edit the path and filename for the output files. The edit box has the control ID 1152, and we can use the same technique that we used for the Open dialog box:

```
IntPtr SaveAsHwnd = getDialog(Hwnd,
    "Save As", 5000);
StringBuilder outfile = new
    StringBuilder(
        @"C:\My Videos\VTS_01_1.n.xxx");
IntPtr SaveEditHwnd =
    GetDlgItem(SaveAsHwnd, 1152);
set = SendMessage(
    new HandleRef(VobEdit,
    SaveEditHwnd),
    WM_SETTEXT,
    IntPtr.Zero, outfile);
IntPtr SaveBtn =
    GetDlgItem(SaveAsHwnd, 1);
SendMessage(new HandleRef(VobEdit,
    SaveAsHwnd),
    WM_COMMAND, new IntPtr(1),
    SaveBtn);
```

As soon as the Save button is pressed, VOBedit will start to de-multiplex the VOB files on the DVD, joining their content together to form one MPEG2 video and AC3 audio file. Detecting when an application has finished is not difficult, but the best method varies from application to application. In the case of VOBedit, a window appears showing the progress made converting the file. Simply testing to see when this window disappears is enough to tell when it has finished, but you could also retrieve the text from the title bar that tells you the percentage of the task that has been completed. To close the application, use the Kill method of the process class or send a message to the Close button on the main form.

This kind of technique can be used to control many applications. You can also intercept messages intended for other windows, which opens up even more possibilities.